Resilience of Stateful IoT Applications in a Dynamic Fog Environment

Umar Ozeer, Xavier Etchevers, Loic Letondeur, François-Gaël Ottogalli, Gwen Salaün, Jean-Marc Vincent

Resilience of Stateful IoT Applications in a Dynamic Fog Environment

Umar Ozeer, Xavier Etchevers, Loïc Letondeur, François-Gaël Ottogalli
Orange Labs
Meylan, France
firstname.lastname@orange.com

Gwen Salaün, Jean-Marc Vincent
Univ. Grenoble Alpes, CNRS, Inria, LIG
Grenoble, France
firstname.lastname@inria.fr

ABSTRACT

Fog computing provides computing, storage and communication resources at the edge of the network, near the physical world. Subsequently, end devices nearing the physical world can have interesting properties such as short delays, responsiveness, optimized communications and privacy. However, these end devices have low stability and are prone to failures. There is consequently a need for failure management protocols for IoT applications in the Fog. The design of such solutions is complex due to the specificities of the environment, i.e., (i) *dynamic infrastructure* where entities join and leave without synchronization, (ii) *high heterogeneity* in terms of functions, communication models, network, processing and storage capabilities, and, (iii) *cyber-physical interactions* which introduce non-deterministic and physical world's space and time dependent events. This paper presents a fault tolerance approach taking into account these three characteristics of the Fog-IoT environment. Fault tolerance is achieved by saving the state of the application in an uncoordinated way. When a failure is detected, notifications are propagated to limit the impact of failures and dynamically reconfigure the application. Data stored during the state saving process are used for recovery, taking into account consistency with respect to the physical world. The approach was validated through practical experiments on a smart home platform.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; **Sensors and actuators**; **Reliability**;

KEYWORDS

Fog Computing; IoT; Fault Tolerance; Heterogeneity; Dynamicity; Cyber-physical Interactions

1 INTRODUCTION

Cloud computing has been, for more than a decade now, an efficient way of renting resources and services to businesses as well as to public users. The outsourcing of applications and services to the cloud is attractive particularly due to its pay-as-you-go model, the potential of delegating the maintenance and storage of physical servers, and the additional services provided like security, elasticity or reliability. However, cloud systems, located within the core network may fail to resolve some of the needs of the Internet of Things (IoT) applications such as low network latencies, QoS and privacy. With the explosion of the number of connected objects and the ever increasing demand for real time processing and data privacy, we are now witnessing the advent of more distributed paradigms to extend the possibilities of the cloud for IoT. *Fog computing* [4] provides resources (compute, storage and communication), and allows control, processing and filtering at the edge of the network, in proximity to the real world. This makes the Fog especially appealing to IoT applications.

Providing reliable services is a key challenge for IoT applications in the Fog. Fault-tolerant services give a better user experience. In some cases, fault tolerance is very important since the non-containment of failures may impact the physical world (PW) as well as provoking the failure of the whole application. These failures may be potentially critical. For instance, the failure of a smoke detector or a lamp in a smart home for elderly/medicated people may be hazardous. Failures in a Fog environment occur regularly because end devices have low stability and are prone to many types of failures like power failures (accidental unplugging, battery drain), hardware failures (due to external environment conditions or wear-out), or software failures. Additionally, end devices are often connected through wireless networks offering both convenience and flexibility to end users. However, it also induces instability due to the volatility of the network and mobility of devices.

Building fault-tolerant systems to provide reliable services for stateful IoT applications in the Fog is challenging as it implies the saving of states, detection of failures and recovery in a consistent manner with respect to the specificities of the Fog-IoT context. For the design of a fault tolerance solution, we consider the following three specificities of the Fog environment: dynamicity, heterogeneity and cyber-physical interactions.

Dynamicity. Devices in the Fog-IoT environment may appear and disappear without any synchronization due to (unannounced) mobility, instability of network links or failures. Failure detection in such an environment is complex because of the entailing uncertainties. It is difficult to know precisely the resources available for the execution and recovery of the application. Dynamicity also includes the support for on-the-fly (re)placement, (re)deployment and reconfiguration of the application. This type of dynamicity can be leveraged for recovery. For instance, the execution of software element can be moved onto other infrastructure entities.

Heterogeneity. The Fog-IoT environment is highly heterogeneous in terms of hardware, software, functions, observability, administrability, network and communication models. The processing architectures (ARM, X86, MCU) and their capabilities are diverse (e.g.: frequencies, cores). The storage properties vary in capacity and in persistency, suggesting that data for recovery may not always be stored locally. The functions of applicative entities may be constrained. This means that not all entities are able to construct or store the data representing their states. Moreover, devices have

different means and degrees of administration and observation. This makes it impossible to rely on just one uniform technique for failure detection, state saving and recovery as certain devices can only be accessible through closed APIs while others can offer a full range of operations to administrate them (e.g.: deployment of softwares, their reconfigurations and lifecycle management in general). Network protocols vary from wired, Wi-Fi, Bluetooth, Z-wave or Zigbee and have their own specificities. This implies extended possibilities for means of observation but also variability regarding fault detection uncertainties. There may be multiple communication models implemented within the same application, like message passing, publish-subscribe, (a)synchronous function calls. This accentuates the need for the co-existence of different techniques for implementing a fault tolerance solution.

Cyber-physical Interactions. The Fog-IoT context includes devices which can frequently interact with the PW. Cyber-physical interaction introduces non-deterministic events that can be PW time, space and context dependent. For instance, the events provoking the turning off of a lamp in an office (e.g.: because of an increase in light intensity at noon) may no longer be valid a few hours later. The PW cannot be rolled back to a previous state. Furthermore, the output devices that interact with the PW can act on the latter in a definitive way. For example, printing a document cannot be reverted. Additionally, care should also be taken when restoring the states of the devices during recovery. For instance, replaying messages may introduce intermediary states that can have undesirable or even dangerous consequences on the PW. Replaying messages on a lamp may cause it to blink multiple times within a time frame and provoke its failure or replaying messages on an injection device of a patient may inject already administered doses of a drug.

This paper proposes a failure management protocol for stateful IoT applications in the Fog taking into account the three specificities mentioned above. The failure management protocol consists of four steps: (i) state saving, (ii) monitoring and failure detection, (iii) failure notification and reconfiguration, and (iv) decision and recovery. We propose a combination of different state saving techniques based on rules and policies to cope with the challenges of the Fog-IoT environment. The state saving policies are based on uncoordinated checkpoint, message log and function call record. A state saving policy indicates the data representing the state of an entity and the corresponding technique of saving. The state saving policy of an entity is governed by its communication model, its functions and the local assumptions that can be made. The protocol monitors infrastructure and applicative entities for failure detection. When a failure is suspected, a decision is taken before engaging into a repair and recovery process. If the suspicion is indeed considered as a failure, a notification is propagated to entities having dependencies on the failed entity which may lead to their reconfiguration in regard to the failure. For the recovery phase, the data stored during the state saving phase are used to restore a consistent state of the application, keeping consistency with respect to the PW (PW-consistency). PW-consistency is ensured by taking into account (i) events that are time sensitive and geographically tied, and (ii) the impact of the technique of recovery on the PW. Dynamicity of the environment is leveraged for recovery: a failed IoT device can be substituted with another functionally equivalent device to ensure the continuation of the functions of the application, and failed software elements can be re-placed and re-deployed on other infrastructure entities before restoring their states.

The contributions of this paper are:

- State saving techniques based on uncoordinated checkpoint, messages log and function call record for stateful IoT applications in the Fog taking into account the specificities of the environment.
- A model of the PW and a mechanism for PW-consistent recovery.
- The design of an end-to-end resilience approach for stateful IoT applications in the Fog, including failure detection, failure notification, on-the-fly reconfiguration and consistent state recovery.
- An evaluation of the failure management solution on a platform-based smart home use case.

The rest of this paper is organized as follows: Section 2 focuses on related work around state saving techniques and more generally on existing failure management systems. The definitions of the infrastructure, application, PW and failure models are given in Section 3. Section 4 aims at describing the failure management protocol. Section 5 presents its implementation and evaluation on a use-case. Section 6 concludes and discusses future work.

2 RELATED WORK

This section is divided into a first part dedicated to checkpoint and log-based (also known as message log) recovery techniques, which are important in distributed systems to achieve fault tolerance. The second part focuses on existing failure management systems and protocols.

2.1 Checkpoint and Message Log

Checkpointing involves saving a set of data representing a state of execution, from which that execution can be resumed, thereby limiting the amount of lost computation after a failure. Checkpoint techniques are often categorized under three main schemes [10, 16]: coordinated, uncoordinated or communication induced.

In *coordinated checkpoint*, also called global checkpoint, the entities of an application synchronize to construct a globally consistent checkpoint [9, 12, 17, 21, 22]. *Uncoordinated checkpoint* [28], on the other hand, removes the synchronization overhead, where the entities construct their checkpoint individually, at the expense of global consistency. In this case, a global consistent checkpoint has to be computed from the available checkpoints. If a global consistent checkpoint cannot be computed, it leads to a domino effect [24]. *Communication-induced checkpoint* [29] allows processes to construct their checkpoint individually while avoiding the domino effect by forcing some entities to construct additional checkpoints. Checkpoint schemes rely on global rollback for consistency restoration which has a system wide impact since all entities, including failure-free entities, have to rollback after each failure.

In a Fog-IoT environment, the synchronization overhead to implement coordinated checkpoint becomes very large as the application scales since all executing entities have to synchronize. Also, the mean time for the rollback recovery can be higher than the mean time between failures resulting in the impossibility to recover. Such an outcome is quite probable in the Fog because as the application scales, the probability of a failure is high and subsequently the mean time between failures (MTBF) is low. IoT devices in the Fog can interact with the PW and may not be able to rollback, for instance,

because they act on the PW in a definitive way. Moreover, after a global rollback, there is no guarantee that the pre-failure execution can be regenerated because of non-deterministic events from the PW. Finally, a globally consistent state within the application may not be consistent with respect to the PW.

Checkpoint can be combined with message log to allow the application to recover beyond the most recent checkpoint. When the entity recovers, it restarts to its most recent checkpoint and the logged messages are replayed causally to reach the pre-failure state. Message log involves saving determinants of non-deterministic events and relies on a piecewise deterministic assumption [2, 25]. By replaying the non-deterministic events in their causal order, the entity can deterministically reach its state prior to the failure. Message logging has three main schemes depending on how determinants are logged [10]: pessimistic, optimistic and causal.

In *pessimistic log*, the determinants are logged synchronously, before they are processed by the recipient. It ensures that before being processed, each message has been successfully logged. *Optimistic log* reduces failure-free overhead by saving locally or in a volatile storage, a set of determinants before flushing them asynchronously in a stable storage. In this case, recovery may, however, be more complex since determinants in the volatile logs are lost when failure occurs. In *causal log* [1], the causal effects of the deliveries of messages are tracked and piggybacked on applicative messages. It combines the advantages of optimistic log while retaining most of the advantages of pessimistic log. It limits the rollback to the most recent checkpoint saved on the stable storage [10].

Uncoordinated checkpoint combined with pessimistic message log is particularly attractive for IoT applications in the Fog. This is because there is little to no computation loss, determinants of non-deterministic events from the PW can be logged and it ensures a minimum disruption in the application during recovery. Unlike global checkpoint, failed entities can recover individually without impacting failure-free entities. Since the storage properties of devices in the Fog differ, uncoordinated checkpoint can also be combined with optimistic or causal log. For instance, a persistent storage avoids a rollback when optimistic logging is implemented since the data for recovery will be still available locally for recovery. However, naively replaying logged messages may still cause inconsistencies because of intermediary states.

2.2 Fault-tolerant Management Systems

The rest of this section focuses on related works on fault-tolerant distributed systems closest to our approach.

In [5, 6], the authors present a reconfiguration protocol applying changes to a set of connected components for transforming a current assembly to a target one given as input. Reconfiguration steps aim at (dis)connecting ports and changing component states (stopped or started). The protocol is robust in the sense that all the steps of this protocol preserve a number of architectural invariants. This protocol is also resistant to failures that may occur during the reconfiguration process. This protocol does not easily scale to IoT applications because the authors assume that all components are hosted on a same VM and a unique centralized manager is in charge of the reconfiguration steps.

[11] presents a self-deployment protocol that was designed to automatically configure cloud applications consisting of a set of software elements to be deployed on different virtual machines. This protocol works in a decentralized way, i.e., there is no need for a centralized server. It also starts the software elements in a certain order, respecting important architectural invariants. This protocol supports virtual machine and network failures, and always succeeds in deploying an application when faced with a finite number of failures. The main difference with our work is that [11] considers stateless applications whereas we focus on consistent state recovery, and consequently the state of the application needs to be stored.

[13] proposes a self-healing approach to handle exceptions in service-based processes and to repair the faulty activities with a model-based approach. More precisely, a set of repair actions is defined in the process model, and repairability of the process is assessed by analyzing the process structure and the available repair actions. When an exception arises during execution, repair plans are generated by taking into account constraints coming from the process structure, dependencies among data, and available repair actions. In [7], the authors present fault-aware management protocols, which permit to model the management behavior of composite cloud applications, by taking into account the possible occurrence of faults suddenly occurring and misbehaving components. This approach also proposes to generate plans for changing the actual configuration of an application for, e.g., recovering an application that is stuck because of a faulted node.

A few recent papers have focused on fault tolerance of IoT applications. [30] provides a fault tolerant approach through virtual service composition. Single service and single device failures are supported by using IoT devices of different modalities as fault tolerant backups for each other. [26] discusses the challenges of fault tolerance in IoT and proposes some potential solutions to consider. It suggests that natural redundancy of functionality across devices within the home, as well as usage scenarios, should be exploited to provide fault tolerance and also discusses the issues of this approach, like incorrect context sensing and actuating of devices. [3] proposes a fault-tolerant platform for smart home applications. It provides fault-tolerant delivery of sensor events and actuation commands in the presence of link loss and network partitions. [14] proposes an IoT-based architecture supporting fault tolerance for healthcare environment. The approach focuses on network fault tolerance which is achieved by backup routing between nodes and advanced service mechanisms to maintain connectivity in case of failing connections. These approaches, however, do not consider the restoration of the state of the application nor the consistency with regards to the PW during the recovery process.

3 MODELS

This section focuses on defining the infrastructure and application models, the model of the PW and the failure model considered.

3.1 Infrastructure and Application Models

The Fog infrastructure is composed of two types of devices: (i) *Servers* which can be administrated to provide computing, storage and communication resources (ii) *Appliances* which provide a dedicated fixed set of services only operable through their exposed

API. *Network Channels* provide the transmission media between a couple of servers or a server and an appliance.

An infrastructure is modeled as a graph, $G_{infra} = (V_{infra}, E_{infra})$. Each vertex represents a Server or an Appliance. Each edge is a Network Channel. The vertices and edges are identified uniquely.

An application is composed of the following entities:

- *Software Elements* are units of software to be executed. They participate in the execution of the application through their corresponding functions. A software element has an internal state and exposes a set of interfaces for its administration.
- *Appliances* are only accessible through their exposed API and cannot be otherwise operated. Since the software and hardware of an appliance are tied, we refer to an appliance as both an infrastructure and applicative entity.
- *Fog Nodes* host the software elements which will be executed on the server. They provide the underlying resources for the execution of the software elements. Each fog node hosts a *Fog Agent* which is a special software element providing an entry point for managing the lifecycle of the locally hosted software elements.
- *Logical Bindings* are abstractions of the communication models which allow a couple of software elements or a software element and an appliance to interact.

An Application is modeled as a directed acyclic graph, $G_{app} = (V_{app}, E_{app})$. Each vertex represents a Software Element or an Appliance. Bindings are represented by edges. Vertices and edges are identified uniquely. The direction of an edge gives the functional dependency between two vertices. If a vertex v_1 *depends* on a vertex v_2,then, v_1 requires v_2 to be functionally operable. In this case, v_2 is said to be *prerequisite* to v_1.

We further assume that events exchanged between the applicative entities are identified uniquely.

3.2 Physical World Model

The physical world is modeled as a finite set of n geographical spaces, $PW = \bigcup_{i=1}^{n} GS_i$. A geographical space is defined as $gs = (id, D, ET)$ where id is a unique identifier, D identifies a three dimensional Euclidean space, ET is a finite set of m couples, $((e_1, t_1), ..., (e_m, t_m))$, representing the state of gs where e is an event sensed or actuated and t is the associated expiration time of the event.

3.3 Instance Model

Figure 1 illustrates an application placed and deployed onto a target infrastructure in a smart home context. The infrastructure entities are composed of two servers, four appliances and two network channels:

- Servers: S1-RPI3 is a Raspberry Pi 3 and S2-PC is a PC.
- Appliances: Motion Sensor, Connected Door Bell, Connected Door Lock and a Camera.
- Network channels: NC-S1S2 is a cabled network channel between the two servers and NC-W-A, a wireless network channel between S2-PC and the appliances.

The application is composed of 2 fog nodes:

- fgn1 hosts the software element MQTT Broker
- fgn2 hosts the software elements CEP (Complex Event Processing) and IoT-Obj-Mgr (IoT Object Manager)

The arrows illustrate the bindings and the functional dependencies. The bindings between the software elements are implemented over the network channel NC-S1S2 whereas the bindings between IoT-Obj-Mgr and the appliances are implemented over the wireless network channel NC-W-A. The porch of the house, where the appliances are located, is represented by the geographical space PW-gs1-porch. The fog agents and their monitoring functions are later illustrated in Figure 5.

3.4 Failure Model

Failures are classified into two categories, namely applicative and infrastructure failures.

Failures of applicative entities are modeled as fail-stop which affect software elements. A software element crashes when it does not execute any further operations. We assume that bindings do not lose messages but can fail by crashing.

Infrastructure failures affect appliances, servers and network channels; these can crash and later recover. The disappearance of an appliance resulting from its crash, the crash of its network channel or an unannounced mobility is considered as a fail-stop of the appliance. A server fails when it can no longer provide the underlying resources to a fog node and will thus induce the failure of the hosted fog node and its software elements. A network channel crashes when it can no longer transmit data in an acceptable time regarding its mean bandwidth. The failure of a network channel can disconnect the server/appliance from the infrastructure, in which case the server/appliance is considered as failed. Figure 2 shows the causality between infrastructure failures and applicative failures: the failure of a network channel induces the failure of the underlying binding(s). The failure of a server results in the failure of the hosted fog node and software elements.

The proposed failure model is motivated by real case failures that can be observed in a Fog-IoT environment as discussed in [20]. A server can fail because of a power failure or overheating of the hardware. The failure of an appliance can, for instance, arise due to a hardware failure because of external environment conditions

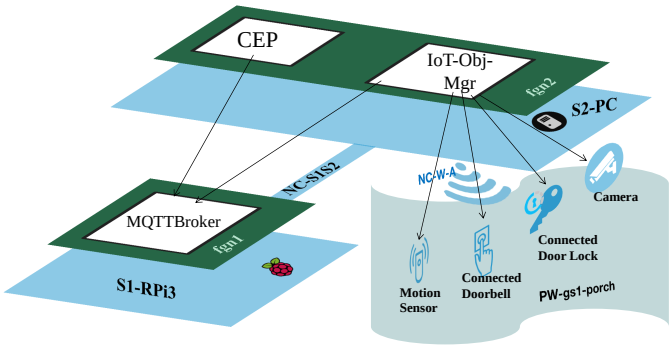

Figure 1: An Instance of an Application on a Fog Infrastructure

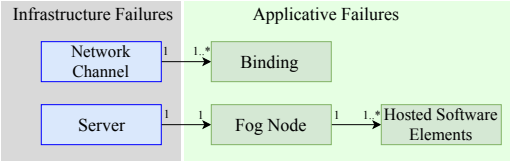

Figure 2: Causality Between Types of Failures

and wear out, or from power failure because of a battery drain. Moreover, infrastructure entities may be connected via wireless network links which can be volatile. Infrastructure entities can, thus, temporarily lose synchronization. Software elements can crash due to a lack of resources for their execution, unhandled exceptions or design/development errors (e.g.: bugs).

4 FAILURE MANAGEMENT PROTOCOL

This section presents the failure management approach. Section 4.1 introduces the failure management entities and their roles. Section 4.2 gives a detailed description of the failure management protocol.

4.1 Failure Management Entities

Figure 3 shows the functionalities of the failure management protocol and the corresponding entities involved in ensuring these functions. The arrows show the relation between the different functions. The participating entities are classified into global and distributed entities. The global entities are:

- A *Stable Storage* which is unaffected by failures. It is available to the software elements to persist data (1.1) (1.2). The implementation of the stable storage may take various forms [18] but for simplicity we assume a reliable dedicated storage infrastructure.
- A *Global Manager, GM*, which is a control and decision making entity that has a global view on the infrastructure and the application. It keeps a record of failed entities. GM is responsible for decision making (3.1) (4.3) after receiving failure suspicions (2.2). It also guides the fog agents during the recovery process (4.3). It retrieves stored states from the storage (4.1) and dependency information from the Application Lifecycle Manager (ALM) (4.2). GM can be hierarchically distributed for managing a set of applications/services within geographical regions. The GM instances can monitor each other and can decide the recovery procedure for failures that impact multiple regions. For instance, the failure of a third party service localized in a region may impact another service running locally in a smart home or a failed service may be recovered in a different region (in the cloud for example). In this paper, GM is treated as a single functional entity.
- The *Application Lifecycle Manager, ALM*, is responsible for the lifecycle operations [20] of infrastructure and applicative entities. For instance, it is involved in the placement and deployment of software elements.

The distributed failure management entities and their roles are:

- *Device Enrollers* are the entry points where IoT devices can be enrolled to participate in the execution of the application. The ALM keeps track of the enrolled devices (0.1).

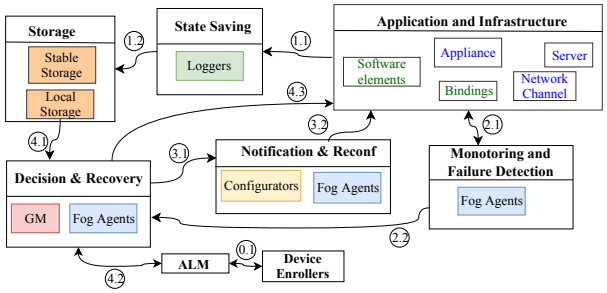

Figure 3: Failure Management Entities and their Functions

- *Fog Agents* have a monitoring responsibility (2.1) which concerns infrastructure entities and software elements hosted on their local fog nodes. They report any failure suspicions to GM (2.2). The fog agents are also involved in the recovery process. They receive the state data from GM for the consistent recovery of software elements and appliances.
- *Software Element Loggers* are implemented as a wrapper over software elements and are responsible for saving the states (1.2) of applicative entities according to their corresponding state saving policies.
- *Configurators* propagate reconfiguration notifications (3.2) when their associated software elements are reconfigured.

The next section discusses the failure management protocol.

4.2 Failure Management Protocol Features

The failure management protocol consists of four steps: (i) State Saving, (ii) Monitoring and Failure Detection, (iii) Failure Notification and Reconfiguration, and, (iv) Decision and Recovery. The following sections describe in more details the four steps of the failure management protocol.

4.2.1 State Saving. The techniques used for saving the state of an entity are policy-based. A state saving policy defines the data to be saved as well as the corresponding technique of saving. The policies are built upon three strategies: uncoordinated checkpoint, message log and function call record. A state saving policy consists of one of these strategies or a combination of two or three strategies.

Uncoordinated checkpoints are performed for both software elements and appliances where data governing the behavior and the execution of the entity like tuning parameters, environment variables and dependencies, composition and configuration files have to be stored. This checkpoint represents an initial state. All checkpoints following this initial state are *delta* checkpoints. A delta checkpoint represents the change in state since the last checkpoint. After a successful checkpoint, all previous checkpoints, message logs and function call records of the entity are purged. A checkpoint is stored locally or on a stable storage depending on the persistency mode of the local storage.

Checkpointing can be combined with message log and/or function call record. Figure 4 illustrates the criteria for choosing between the different state saving policies. The policies are defined with reference to the properties of the local storage, the communication model, and the dependencies on other applicative entities. The

last change of state (or event changing state) should be recorded. The nature of the local storage determines between optimistic and pessimistic techniques. A volatile local storage implies that the locally saved data can be lost when there is a failure. Thus, saving is done in a pessimistic way ensuring that all the required data for consistent state recovery is available. A persistent local storage, on the other hand, can support optimistic techniques since the data can still be available for recovery even in case of failures. The implemented communication model between two interacting entities also impacts the choice of the saving policy. Message log is implemented for message based communication and function call record for a model based on function calls. Message log or function call record is generally performed at reception[1] since reception is non-deterministic: this is the case when the emitter and receiver are both software elements or an appliance and a software element, respectively. However, when the receiving entity is an appliance, message log or function call record is performed at emission since appliances cannot be expected to save their state. To achieve message log and function call record, a logger is wrapped around software elements. The state saving of appliances are achieved by software elements loggers since appliances have constrained functional perimeters and cannot save their state. The message logger intercepts the reception or emission (depending on the policy) of messages and logs them before delivering the message to the software element for processing. The function call logger intercepts function calls and records the call before calling the original function itself. The checkpoint frequency is defined, based on execution time or on the number of processed events.

Moreover, an event, (*evt*), sensed or pushed for actuation in the PW, is saved by enriching it with a *recovery validity timer (rvt)*, which is piggybacked on applicative messages to track its causality in the application. The recovery validity timer gives the number of seconds for which the event is valid in the PW after its occurrence. A set of couples (evt, rvt) constitutes the state of a geographical space in the PW. The *rvt* value depends on the PW context in which an event was sensed or pushed for actuation and the time frame for which the resulting state with regards to the PW should be maintained. The expiration of the *rvt* value means that this event is no longer consistent with the PW and should not be taken into account during a recovery phase. An event can have an immediate expiration time or can invalidate the expiration timer of a previous event. For instance, temperature events sent by a connected thermometer every second may have *rvt=1*; a message to unlock a connected door lock may have *rvt=10* but may be overridden by a message to lock the door.

4.2.2 Monitoring and Failure Detection. Fog agents have an active monitoring role. They monitor both infrastructure and applicative entities:

- *Local Software Elements*: The fog agents monitor all the software elements running on their local fog nodes. They report any failure suspicions of software elements to the global manager.
- *Infrastructure Entities*: The fog agents also monitor remote servers and appliances. These remote entities to monitor are assigned by the ALM based on geographic proximity.

[1]Function call recording can be achieved at the caller or at the callee. We refer to the former as emission and to the latter as reception, respectively.

Local Storage		Communication Model: Function Call	Communication Model: Messaging	Destination Entity
Persistent		Optimistic Function Call Record at Reception	Optimistic Message Log at Reception	Software Element
		Optimistic Function Call Record at Emission & Reception	Optimistic Message Log at Reception	Appliance
Volatile		Pessimistic Function Call Record at Reception	Pessimistic Message Log at Reception	Software Element
		Pessimistic Function Call Record at Emission & Reception	Pessimistic Message Log at Emission & Reception	Appliance

Figure 4: State Saving Policies

Figure 5 illustrates the monitoring of infrastructure entities and local software elements by the fog agent for the use case presented in Figure 1.

When a fog agent suspects the failure of an entity, it adds the latter to the local list of suspected entities and reports it to the global manager. The fog agent implements four types of monitoring for failure detection depending on the entity to observe:

(1) *Heartbeats*: The fog agent implements a heartbeat mechanism for the monitoring of neighboring servers. It emits a heartbeat towards the fog agent on the neighboring server at regular intervals. It triggers a timeout for the reception of a heartbeat. At each reception of a heartbeat within the timeout, it resets the timeout. If it fails to receive the heartbeat before the expiration of the timeout, the fog agent suspects the failure of the server.
(2) *Applicative messages observation*: For the monitoring of appliances, applicative messages observation is preferred. The fog agent observes the applicative messages of the appliance if the latter communicates at regular intervals (for example a connected thermometer that reports the temperature every second). In this way there is no influence on the monitored appliance by the fog agent. The fog agent suspects the failure of the appliance if it fails to observe a message in the required interval.
(3) *Ping-acks*: The fog agent can also monitor appliances through ping-acks if applicative message observation is not possible (for example if the appliance does not communicate at regular intervals). In this case, the fog agent starts a timeout and sends a message to the appliance requesting a reply. If the appliance has not replied within the expiration of the timeout, the fog agent suspects the failure of the appliance.
(4) *Local System Observation*: The fog agent relies on the local (operating) system observation to monitor local software elements [19]. This avoids influence on the network traffic and message delays will not cause false detection. It also avoids interference with the execution of software elements.

In order to avoid a wrong failure suspicion of a server, the fog agents may recover themselves from unexpected failures [15]. To do so, when the fog agent is initialized, it forks a local backup which keeps polling the primary fog agent. If the primary fog agent fails,

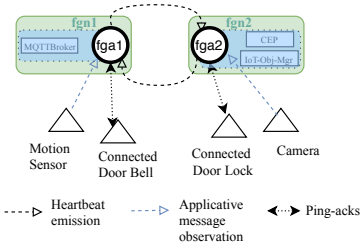

Figure 5: Monitoring of Remote Infrastructure Entities

the backup fog agent breaks the polling loop and resumes the task of the primary fog agent by becoming itself the primary fog agent. It then spawns a new backup. If the backup fog agent fails, the primary fog agent receives a signal (since the backup is a child process) to create a new backup.

4.2.3 Failure Notification and Reconfiguration. The failure of an entity may impact other failure-free entities, for instance because of functional dependencies between them. Failure and reconfiguration notifications aim at limiting service interruption. When a failure occurs, a failure notification is sent to the set of functionally dependent software elements, $fd\_se \in FD\_SE$. The failure notification indicates the entity, fe, that has failed and a *decision* with respect to that failure, so that the following reconfigurations are triggered:
(1) fd_se stops processing events to and from fe,
(2) fd_se implements the decision included in the failure notification. The decision is taken by GM as follows: (a) If fe is a software element, fd_se waits for a recovery notification to start processing events to and from fe again, (b) If fe is a device that interacts with the PW, fd_se is connected to a functionally equivalent device (Cf. 4.2.4), and, (c) If all dependent and prerequisite entities of fd_se have failed, its execution is temporarily stopped.

The reconfiguration of the software element, fd_se, is then propagated. Algorithm 1 describes how these notifications are pushed to implement reconfigurations. In an initialization process, each software element's configurator, se_conf, subscribes to the reconfiguration event on a set of relevant software elements, se, given by the ALM. When se_conf receives a reconfiguration subscription message, it adds the software element as a follower in a local subscription list $ReconfSeFollowersList$. When an entity fails, GM retrieves (from the ALM) its set of functionally dependent software elements, FD_SE and sends them a failure notification and the decision regarding this failure. When se_conf receives a failure notification, it reconfigures its associated software element accordingly and propagates a reconfiguration notification to the set of software elements listed in $ReconfSeFollowersList$. When se_conf receives a reconfiguration notification, it reconfigures its associated software element according to 1, 2(a) and 2(c).

4.2.4 Decision and Recovery. The recovery process is achieved in two steps. The first step is a decision process computed by GM to decide the rules for state restoration. In a second step, the fault management entities, described in Section 4.1, implement the state restoration according to the rules decided in the first step. The decision for state restoration is based on a set of three rules. The first rule depends on whether the failed entity has interactions with

Algorithm 1: Failure Notification and Reconfiguration

```
Input: Set of software elements in the application SE
// Initialisation
foreach se ∈ SE do
    ReconfSeFollowersList = []
    ReconfSeToFollowList = ALM.getDependencies(se)
    foreach seToFollow ∈ ReconfSeToFollowList do
        se.sendReconfSubscription(seToFollow)
    end
end
/* Reception of reconfiguration subscription */
Function ReceiveReconfSub(seFollower):
    logMsg()
    ReconfSeFollowersList.add(seFollower)
    return;
/* Reception of failure notification from GM */
Function ReceiveFailureNotif(fe, decision):
    logMsg()
    localReconf(fe, decision)
    foreach seFollower ∈ ReconfSeFollowersList do
        sendReconfNotification(seFollower, fe, decision)
    end
    return;
/* Reception of a reconfiguration notification */
Function ReceiveReconfNotif(seFollowing, fe,
 reconf):
    logMsg()
    reconf = localReconf(seFollowing, fe, reconf)
    foreach seFollower ∈ reconfSeFollowersList do
        sendReconfNotification(seFollower, fe, reconf)
    end
    return;
```

the PW. Second, the state restoration is based on the state saving policy of the failed entity. The third rule depends on the type of entity that has failed.

Interaction with the PW. The replay of messages and recall of functions produce intermediary states until the final state is reached. For entities interacting with the PW, these intermediary states can cause inconsistencies when recovering. Furthermore, this final state may no longer have any sense in the PW. To overcome these inconsistencies with the PW, the final state of the entity should be computed from its set of message logs and function call records taking into account their rvt value. The computation of the final state requires a functional specification of entities interacting with the PW and how events change their states. A recovery is PW-consistent if, $\forall gs \in PW, E_{rec} = (E_{pf} \backslash E_{exp}) \bigcup E_{rpf}$, where gs is a geographical space, E_{rec} is the set of couples (evt, rvt) representing the state of gs after recovery, E_{pf} is the pre-failure state of gs, E_{exp} is the set of events having an expired rvt, and E_{rpf} is a set of events sensed and actuated over gs after the occurrence of the failure but before recovery.

The state saving policy. Given a set of message logs (function call records, respectively) of a failed entity, consistency restoration involves replaying message logs (recalling the functions recorded, respectively) after restoring the state of the entity with a checkpoint if one is available. A message log (function call record, respectively) with an expired *rvt* value is not replayed (recalled, respectively) during the recovery process.

Nature of the failed entity. The rules for recovery differ for appliance, software element and server failures.

Appliance failure. An appliance failure leads into two possibilities of consistency restoration: i) replacement, or ii) degraded mode. Replacement is a situation where an alternative, functionally[2] equivalent appliance is used for consistency restoration. In this case, the state of the failed appliance is restored on a different appliance. Degraded mode applies when no alternative appliance can replace the functions of the failed one. In this case, the application continues its execution with less features. Note that, if the appliance is essential to the functions of the application, for instance if all other entities depend on the appliance, the whole application fails.

Software element failure. To recover from a software element failure, the latter is re-instantiated on the same fog node. If re-instantiation on the same fog node fails (for example because of a lack of resources for its execution), a new placement is computed for the software element. A placement [27] request to the ALM, which knows its physical and logical constraints, returns a new possible fog node for the execution of the software element. The ALM re-deploys the software element, then the state of the latter is restored before resuming its execution. A reconfiguration notification is then propagated as described in Section 4.2.3

Server failure. Recovery from a server failure is complex since it causes the failure of its fog node and all the hosted software elements. It also means that the application has now fewer resources for its execution. The software elements are re-placed/deployed (by the ALM) before restoring their states. The impossibility of re-placing all the software elements may lead to a degraded mode (execution with a fewer features) or a failure of the whole application, i.e., the impossibility to recover if the software elements are essential to the function of the application.

The fog agent has also an active responsibility in state restoration of software elements and appliances. After determining the rules for state and consistency restoration, GM retrieves the state data of a failed entity from the storage. It provides the data as well as the rules for state restoration of the failed entity to the associated fog agent. After a successful recovery, GM sends a recovery notification to $fd\_se \in FD\_SE$, which triggers the reconfiguration of the application according to Algorithm 1.

5 EVALUATION

This section gives an evaluation of the resilience protocol and aims at answering the following questions: (i) Is recovery of the application successful when a failure occurs? (ii) Is the failure management protocol able to repair different types of failures within an acceptable delay from a user point of view? (iii) Is the recovery process consistent with respect to the PW?

[2] An appliance a_1 is functionally equivalent to another appliance a_2, if a_1 can provide at least the same functionalities as a_2. Functional equivalence is given by the Thing'in platform developed by Orange. It provides a digital index of connected things and their relationships. The platform can be interrogated for devices providing required services and having specific constraints. More information on this on going study can be found online at http://thinginthefuture.com/

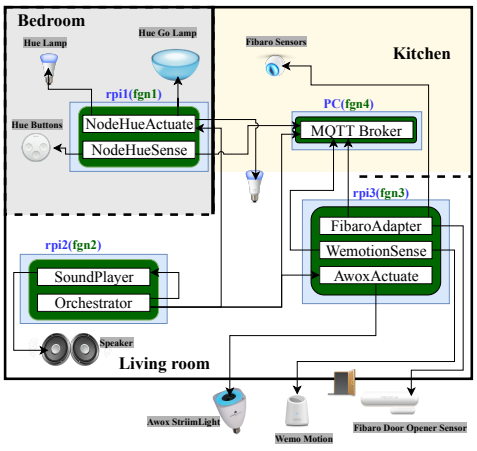

Figure 6: Smart Home Application and Infrastructure

5.1 Use Case Description

The target use case application is a smart home where, for comfort and convenience of the house tenants, available devices are remotely controlled for automated lightning and physical intrusion detection. Figure 6 depicts the smart home application deployed on the target infrastructure. The infrastructure and application were chosen such that they represent legitimate entities that can be found in real-life smart home [20] and include entities with the three specificities of the Fog-IoT environment discussed in Section 1.

The infrastructure is composed of three Raspberry Pi (Model 3 Type B) and a PC. Software elements are hosted on four fog nodes: *fgn1 (NodeHueSense, NodeHueActuate)*, *fgn2 (Orchestrator, Soundplayer)* and *fgn3 (FibaroAdapter, AwoxActuate)* hosted on three distinct Raspberry Pi and *fgn4 (MQTT Broker)* hosted on a PC. The appliances are: Philips Hue Lamps, Hue Go Lamp and Hue Buttons, an Awox StriimLight (connected in WiFi), Fibaro Door Sensor, Fibaro Sensors (which reports motion, light intensity, vibration and temperature), a WeMo Motion Sensor and a Speaker.

The applicative entities and their functions are:

- A Message Oriented Middleware (MOM) that allows a publish-subscribe communication pattern. It implements a MQTT broker based on ActiveMQ.
- The *Orchestrator* defines the actions that should be triggered based on events reported by sensors.
- *NodeHueSense* reports events from Hue Buttons while *NodeHueActuate* controls the Hue Lamps.
- *FibaroAdapter* reports events sensed by the Fibaro Sensors and publishes them on the MQTT broker.
- *WemotionSense* reports motion events sensed by the Wemo Motion Sensor and publishes them on the MQTT broker.
- *AwoxActuate* controls the light and the integrated speaker of the Awox StriimLight, and *SoundPlayer* controls the Speaker in the living room.

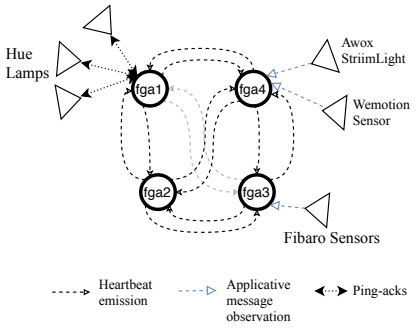

Figure 7: Monitoring of the Infrastructure by Fog Agents

5.2 Failure Management Framework and Setup

We developed a resilience framework which provides four APIs to extend the functions of software elements: (i) *CkptAPI* defines the data and the frequency of the uncoordinated checkpoints, (ii) *MsgLogAPI* intercepts messages and handles their logging according to the policy defined, (iii) *FctRecordAPI* overrides the functions implemented by a software element to handle the recording of function calls, and, (iv) $ConfigAPI$ handles the reconfiguration of software elements.

A monitor is deployed on each fog node as part of the fog agent's functions. The monitor is capable of failure detection through the mechanisms described in Section 4.2.2. Each applicative and infrastructure entity to be observed is associated to at least a fog agent. Its parameters (frequencies, timeouts) can be tuned according to the use case application, network latencies and the desired reactivity. GM has to be deployed on a reliable infrastructure. The stable storage takes form of a *MongoDB* database. The framework was developed in *Node.js* because it is lightweight, allows asynchronous operations and its packet manager, *npm*, handles effectively the management of runtime dependencies.

The resilience framework is deployed onto the use case application. GM and the stable storage are deployed on a dedicated reliable laptop which is unaffected by failures. A fog agent is deployed on each fog node. Figure 7 illustrates the monitoring of this use case infrastructure with the fog agents. The frequency of heartbeats is set to 500ms to account for network latencies. The appliances to monitor are assigned to the respective fog agents according to their location and observability. The fog agents monitor the local software elements though the local system observation at a frequency of 100ms.

5.3 Experimental Results

We experiment our resilience approach through different types of failures so that the different recovery rules described in Section 4.2 can be illustrated and evaluated. We designed three tools for evaluation purposes: (i) a *Scenario Injector* generates a set of sensor events and injects them into the application to change its state, (ii) a *Failure Injector* provokes the failure of appliances, software elements or servers, and, (iii) a *Verifier* checks that the recovery is consistent with respect to the PW.

We designed three experiments for the evaluation of the resilience protocol. In a first experiment, we observe the behavior of the failure management protocol for software element failures. We set the failure injector to provoke the failure of random software elements and measure the time for their re-instantiation on the same fog node. The experiment is carried out with one to four simultaneous injected failures. The experiment is repeated one hundred times with an interval of five seconds between failures. Figure 8 illustrates the results of the time measured for repair with respect to the number of failures. It takes around 100ms to repair the failure of one software element. In the case of four simultaneous failures, in more than 75% of cases it takes less than 300ms to repair. Expectedly, the median values for the time to repair increases with the number of failures. This is because GM has to process multiple failure suspicion notifications and the fog agents have to handle the repair of multiple local software elements. The variability of the measured values for multiple failures is mainly due to: (i) the time for the fog agents to process multiple messages from GM, and, (ii) the time taken for the software elements to restart.

In a second experiment, we measure the time taken to repair the application when a server fails. We measure the time to re-instantiate the software elements hosted on $fgn1$ on a different fog node (e.g.: *fgn2*) and reconfigure the application when *rpi1* fails. Figure 9 shows that in more than 75% of cases, it takes between 3.5s and 4.2s to repair and reconfigure the application in this case. This time is, expectedly, higher than the time of re-instantiation on the same fog node, since the time for the detection of the failure is higher as the frequency of heartbeats is lower than the frequency of observation of software elements. Moreover, GM has to send messages to *fga2* for the observation of the newly instantiated software elements and subsequently wait for acknowledgements of their successful restart.

The third experiment aims at showing the reconfiguration and the consistent recovery with respect to the PW. The scenario injector is used to input random events to turn on or off the Hue Lamp in the bedroom. When the failure of the lamp is provoked, the application is reconfigured so that the bedside Hue Go Lamp is used to function as the main lamp of the bedroom. The state of the failed lamp is restored on the bedside Hue Go Lamp to keep PW-consistency. Figure 10 illustrates the time taken for the state restoration for one hundred failures at five seconds time intervals. This time is between 10ms and 20ms in more than 75% of cases.

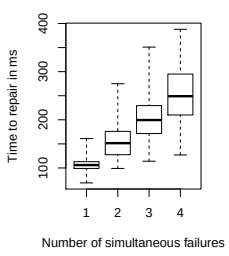

Figure 8: Time to Re-instantiate Failed Software Elements on the Same Fog Node

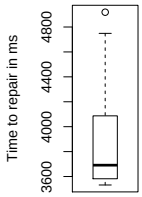

Figure 9: Time to Repair Software Elements Hosted on *fgn1* when *rpi1* Fails

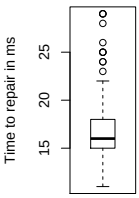

Figure 10: State Restoration on a Lamp

This section evaluated the behavior of the failure management protocol with respect to the time taken to repair infrastructure and applicative entities. The experiments show that the resilience protocol works well in practice, even in case of multiple failures. The repair of software elements on the same fog node is performed within a short delay from the user point of view. According to [8, 23] one second is the limit of response time for the user's flow to stay uninterrupted even if the delay is noticeable. These experiments also show the benefit of re-instantiation of software elements on a same fog node rather than moving to another fog node. The repair of server failures is performed within an acceptable delay in this context since the frequency of failures of servers is intrinsically lower than the the frequency of software elements (because of the causality relation between the two types of failures). Moreover, consistent state restoration is achieved within a very satisfactory delay. In this case, the resilience protocol offers comfort and convenience to house tenant since there is PW-consistent recovery and an automatic switch to the bedside lamp.

6 CONCLUSION AND FUTURE WORK

This paper presented a fault tolerance solution for stateful IoT applications in a dynamical Fog environment, taking into account the specificities of the environment, namely dynamicity, heterogeneity and cyber-physical interactions. Fault tolerance is achieved by saving the state of applicative entities. The failure management protocol monitors both infrastructure and applicative entities for failure detection. When a failure is detected, the application is reconfigured and the data stored during the state saving phase are used for a consistent state recovery, including PW-consistency. An evaluation procedure of the failure management protocol is proposed. The evaluation shows that the protocol is robust and is able to restore the application in a consistent and stable state in the presence of multiple simultaneous failures. The experiments show that the protocol is able to recover from failures in a reasonable user time.

Future works include: (i) an extensive performance evaluation of the failure management protocol, (ii) improving the techniques for better monitoring of appliances for failure detection, and (iii) extension of the failure management protocol for non-administrable software elements (black-boxes).

REFERENCES

[1] L. Alvisi, K. Bhatia, and K. Marzullo. 2002. Causality Tracking in Causal Message-logging Protocols. *Distrib. Comput.* 15, 1 (2002), 1–15.

[2] L. Alvisi and K. Marzullo. 1998. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Trans. on Software Engineering* 24, 2 (1998), 149–159.

[3] M. S. Ardekani, R. P. Singh, N. Agrawal, D. B. Terry, and R. O. Suminto. 2017. Rivulet: A Fault-tolerant Platform for Smart-home Applications. In *Proc. of Middleware'17 (Middleware '17)*. ACM, 41–54.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. 2012. Fog Computing and its Role in the Internet of Things. In *Proc. of MCC'12*. ACM, 13–16.

[5] F. Boyer, O. Gruber, and D. Pous. 2013. Robust Reconfigurations of Component Assemblies. In *Proc. of ICSE'13*. IEEE Press, 13–22.

[6] F. Boyer, O. Gruber, and G. Salaün. 2011. Specifying and Verifying the Synergy Reconfiguration Protocol with LOTOS NT and CADP. In *Proc. of FM'11 (LNCS)*, Vol. 6664. Springer-Verlag, 103–117.

[7] A. Brogi, A. Canciani, and J. Soldani. 2018. Fault-Aware Management Protocols for Multi-Component Applications. *Journal of Systems and Software* 139 (2018), 189–210.

[8] S. K. Card, G. G. Robertson, and J. D. Mackinlay. 1991. The Information Visualizer, an Information Workspace. In *Proc. of CHI '91*. ACM, 181–186.

[9] K. M. Chandy and L. Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75.

[10] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. 2002. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.* 34, 3 (2002), 375–408.

[11] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. D. Palma. 2017. Reliable Self-deployment of Distributed Cloud Applications. *Softw., Pract. Exper.* 47, 1 (2017), 3–20.

[12] C. J. Fidge. 1988. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proc. of the 11th Australian Computer Science Conference*. 56–66.

[13] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni. 2010. Exception Handling for Repair in Service-Based Processes. *IEEE Trans. Software Eng.* 36, 2 (2010), 198–215.

[14] T. N. Gia, A.-M. Rahmani, T. Westerlund, P. Liljeberg, and H. Tenhunen. 2015. Fault Tolerant and Scalable IoT-Based Architecture for Health Monitoring. In *IEEE SAS*. IEEE, 1–6.

[15] Y. Huang and C. Kintala. 1995. Software Fault Tolerance in the Application Layer. *Software fault tolerance* 3 (1995), 231–248.

[16] A. Khunteta and P. Kumar. 2010. An Analysis of Checkpointing Algorithms for Distributed Mobile Systems. *Int. Journal on Computer Sci. and Eng.* 2 (2010), 1314–1326.

[17] T. H. Lai and T. H. Yang. 1987. On Distributed Snapshots. *Inform. Process. Lett.* 25, 3 (1987), 153–158.

[18] B. Lampson and H. E. Sturgis. 1979. *Crash Recovery in a Distributed Data Storage System.* Technical Report. Xerox Palo Alto Research Center.

[19] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. 2011. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *Proc. of SOSP '11*. ACM, 279–294.

[20] L. Letondeur, F.-G. Ottogalli, and T. Coupaye. 2017. A Demo of Application Lifecycle Management for IoT Collaborative Neighborhood in the Fog. In *IEEE Fog World Congress*. IEEE, 1–6.

[21] F. Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.

[22] F. Mattern. 1993. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *J. Parallel and Distrib. Comput.* 18, 4 (1993), 423–434.

[23] R. B. Miller. 1968. Response Time in Man-Computer Conversational Transactions. In *Proc. of AFIPS '68 (Fall, part I)*. ACM, 267–277.

[24] B. Randell. 1975. System Structure for Software Fault Tolerance. *SIGPLAN Not.* 10, 6 (1975), 437–449.

[25] R. Strom and S. Yemini. 1985. Optimistic Recovery in Distributed Systems. *ACM Trans. Comput. Syst.* 3, 3 (1985), 204–226.

[26] D. Terry. 2016. Toward a New Approach to IoT Fault Tolerance. *Computer* 49, 8 (2016), 80–83.

[27] Y. Xia, X. Etchevers, L. Letondeur, T. Coupaye, and F. Desprez. 2018. Combining Hardware Nodes and Software Components Ordering-based Heuristics for Optimizing the Placement of Distributed IoT Applications in the Fog. In *Proc. of SAC'18*. ACM, 751–760.

[28] J. Xu and R. H. Netzer. 1993. Adaptive Independent Checkpointing for Reducing Rollback Propagation. In *Proc.* 5^{th} *IEEE SPDP*. IEEE, 754–761.

[29] F. Zambonelli. 1998. On the Effectiveness of Distributed Checkpoint Algorithms for Domino-Free Recovery. In *Proc. of HPDC'98*. IEEE, 124–131.

[30] S. Zhou, K.-J. Lin, J. Na, C.-C. Chuang, and C.-S. Shih. 2015. Supporting Service Adaptation in Fault Tolerant Internet of Things. In *Proc. of SOCA '15*. IEEE, 65–72.